SOFTWARE ENGINEERING LABORATORY SERIES

SEL-87-004

ASSESSING THE ADA® DESIGN PROCESS AND ITS IMPLICATIONS: A CASE STUDY

JULY 1987

NASA

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-87-004

ASSESSING THE ADA® DESIGN PROCESS AND ITS IMPLICATIONS: A CASE STUDY

JULY 1987

NASA

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory(SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC, Systems Development Branch
University of Maryland, Computer Sciences Department
Computer Sciences Corporation, Flight Systems Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The primary authors of this document are

Sara Godfrey (Goddard Space Flight Center)
Carolyn Brophy (University of Maryland)

Other contributors include

William Agresti (Computer Sciences Corporation)
Edwin Seidewitz, Michael Stark (Goddard Space Flight Center)
GRODY Design Team

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

ABSTRACT

The results of a case study to analyze the approach taken and the lessons learned during the design of the Gamma Ray Observatory Dynamics Simulator in Ada[1] (GRODY) are presented. Included are recommendations for defining the design phase and outlining the products that should be developed during this phase of the software development life cycle for future flight dynamics software systems developed in Ada.

[1]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont'd)

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

EXECUTIVE SUMMARY

During the past 2 years, a study has been conducted to determine the applicability of Ada for software development in the flight dynamics environment at the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC), Greenbelt, Maryland. The primary objectives of this study are to determine the cost-effectiveness and feasibility of using Ada and to assess the effect of Ada on the flight dynamics environment. The study consists of parallel software development efforts to develop the Gamma Ray Observatory Dynamics Simulator, with one team of developers using FORTRAN and another team using Ada. A third team collects and assesses data from the two development efforts. The study is a joint project with participants from NASA/GSFC, Computer Sciences Corporation, and the University of Maryland.

This document concentrates on the design phase of the development effort, during which the following conclusions were reached concerning the use of Ada as a development language for flight dynamics applications:

- Training is essential, not only training in Ada, but also in the design methodologies applicable for Ada. Managers and reviewers also need some training in design methodology.
- The specifications document should be language neutral and should not constrain the design.
- The design methodology should be chosen as early as possible and should be suitable for expressing Ada features. During this study, object diagram methodology seemed to be extremely useful.

- Documenting an Ada design requires different design products than are used to describe a FORTRAN design. Object diagrams and compilable design elements seemed very useful for representing and validating the design.
- Designing with Ada seems to require a longer design phase, with reviews occurring later in an Ada design phase than they would in a FORTRAN design phase.
- The changeover to Ada will increase design cost due to the loss of the previous design legacy.

Further study is continuing to assess the effect of using Ada during other life-cycle phases and throughout the development life cycle of other projects.

SECTION 1 - INTRODUCTION AND BACKGROUND

This document is the second in a planned series of five documents describing various aspects of developing a dynamics simulator to be used as part of the ground support system for the Gamma Ray Observatory (GRO) satellite. This project, the Gamma Ray Observatory Dynamics Simulator in Ada (GRODY), is significant because a corresponding version is being developed in FORTRAN. Analysis of the two projects will provide insight into the implications of developing flight dynamics software in Ada versus FORTRAN--the usual development language in the past. This document concentrates on the design phase of the project.

Section 1 provides background material on GRODY, dynamics simulators in general, and the phases of the software development life cycle. Section 2 details the approach taken during the design process. It describes the major activities that occurred during the GRODY design effort, the methodologies investigated, the design documentation, and the staffing profile for the design effort. Section 3 discusses the success of various aspects of the design approach and offers some insight into possible changes in this approach. Section 4 briefly summarizes the lessons learned in the design process and presents some recommendations for conducting the design effort in future Ada development projects.

1.1 GRODY PROJECT DESCRIPTION

The GRODY project is an experiment in the effectiveness of Ada for flight dynamics software development. The experiment, which is being conducted at the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC), features the parallel development of software in FORTRAN and Ada. Supporting this Ada experiment are participants from three distinct installations: NASA/GSFC,

Computer Sciences Corporation (CSC), and the University of Maryland. These people are divided into three functional groups: the FORTRAN development team, which is developing a GRO dynamics simulator under the standard methodology used in the flight dynamics environment (References 1 and 2); the Ada development team, which is applying modified design techniques (References 3 and 4) and using the Ada development language; and a study group, which is directing the experiments. Data on both projects are being collected and stored on the Software Engineering Laboratory (SEL) database. Figure 1-1 illustrates this organizational structure. Work on both dynamics simulators began in January 1985, with the FORTRAN team beginning a typical development cycle and the Ada development team, a training phase.

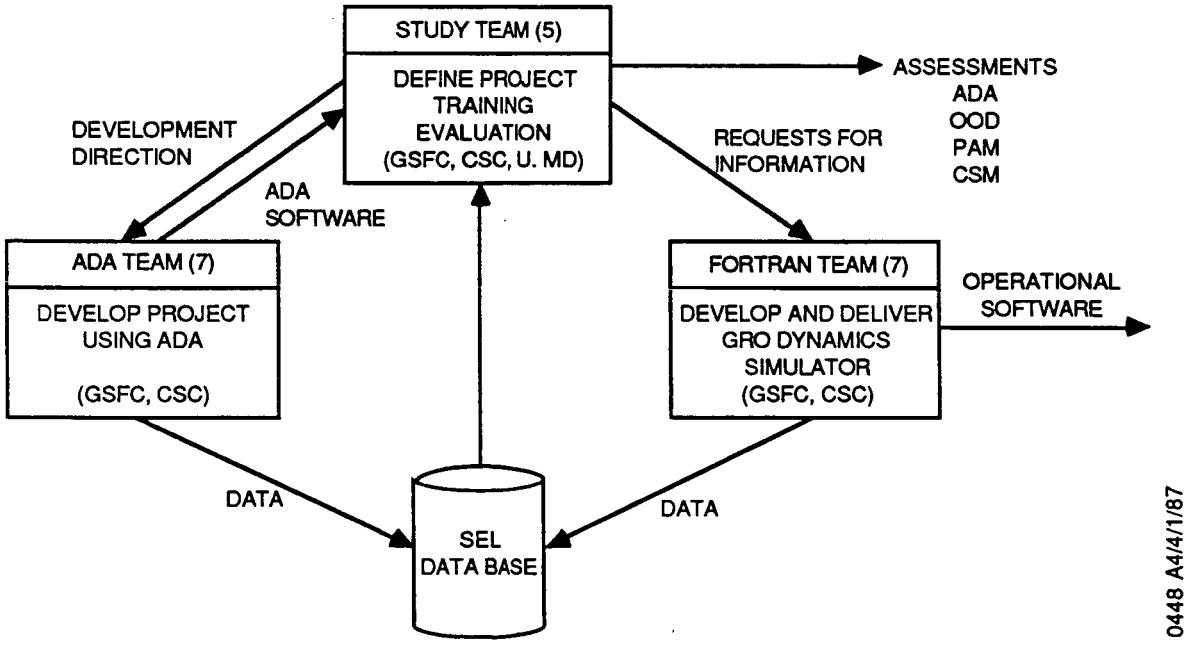


Figure 1-1. Ada Experiment Organization

Because the FORTRAN development team is responsible for the actual software that will be used for GRO mission support,

they are more constrained by their schedule than the Ada team. An additional requirement for the FORTRAN team is to develop one portion of the software so that it can later be integrated into a real-time piece of software to be used for simulation purposes. With these two exceptions, the two teams are developing functionally identical software systems.

The FORTRAN development effort is being carried out using a DEC VAX-11/780 computer; the Ada development is being done on a DEC VAX-8600. Early estimates of the size of the project indicated that the completed system (for both the FORTRAN and the Ada developments) was expected to be approximately 40,000 source lines[1] of code (SLOC). Later estimates for the Ada development place the system size in the range of 110,000 to 120,000 SLOC. Estimates for executable SLOC in the Ada system are 40,000 to 45,000; the completed FORTRAN system contains 25,000 executable SLOC. Table 1-1 lists size comparison information for the two projects. Additional information on the experiment is presented in Reference 3.

Table 1-1. Project Size Comparisons

COMPONENT MEASURED	FORTRAN	ADA[a]
TOTAL LINES OF CODE	45,000 SLOC	110,000 SLOC
TOTAL COMMENTS	19,000 SLOC	43,000 SLOC
EXECUTABLE LINES	25,000 SLOC	40,000 SLOC
BLANK LINES	MINIMAL	27,000 SLOC
REUSED LINES	16,000 SLOC(36%)	2,000 SLOC(2%)

0448 A4/4/6/87

[a] ESTIMATED MAY 1, 1987 (BASED ON CODE 80% COMPLETE).

[1] A source line of code is an 80-byte record processible by the computer. It therefore includes comments, executable code, and nonexecutable code such as type statements, block data statements, and dimension statements for FORTRAN or declarations and blank lines for Ada.

1.2 GRODY OBJECTIVES

The overall goal of the Ada/FORTRAN software development project is to develop insight into the applicability of the Ada development methodology and language in the NASA software environment. Several objectives have been established as a mechanism toward attaining this goal (Reference 3). The primary one is to determine the cost-effectiveness and feasibility of using Ada to develop flight dynamics software and to assess the effect of Ada on the flight dynamics environment. A related objective is to determine whether present development methodologies in use within the flight dynamics environment are suitable for Ada as is or whether they need to be adapted for Ada and to investigate other methodologies related to the use of Ada. For example, is the standard development life cycle (Section 1.5) being used on the FORTRAN development equally suitable for an Ada development? Which design methodology is best suited for this type of Ada development?

Because reusability is an important factor for cost-effective software development, this experiment will also try to develop approaches for maximum reusability when Ada is being used for implementation. A major portion of the software developed in the flight dynamics environment is reused; because Ada is designed to facilitate reusability, the methods developed should maximize this feature.

Other factors being assessed throughout the GRODY project are the differences in reliability and maintainability between a FORTRAN implementation and one in Ada. Obviously, a system that is more reliable (i.e., has fewer errors per 100,000 SLOC) will cost less to maintain.[1] Similarly, an

[1]Software maintenance consists of two activities occurring after the software is delivered: correction of errors discovered during operational use of the software and enhancements of the software to add new capability.

implementation that is easier to correct and enhance will be less costly to maintain. Will it be less costly to maintain the Ada or the FORTRAN dynamics simulator? Which will be more reliable? These questions are particularly important, considering that the annual cost of maintenance ranges from 10 to 35 percent of the original development cost in staff-hours (Reference 2).

Ada is required as the implementation language for the Space Station, an extremely large, complex, long-term project. One objective of the GRODY experiment is to develop a set of software measures that will be helpful when planning for the use of Ada during the Space Station project. Among these measures are size estimates of an Ada implementation and the expected productivity during the implementation of a scientific application. Useful information can also be gained about the reliability and maintainability--the effort required to change and repair the software--when Ada has been used for the implementation. Most of these software measures are well defined for FORTRAN implementations, but not much similar information exists for scientific applications in Ada.

Another factor of interest with the longer term projects such as the Space Station is the issue of portability. Can a particular implementation be moved from one machine to another with ease? Certainly, on long-term projects, the rehosting of the software to a new computer system is a likely occurrence, and a more portable implementation would reduce these rehosting costs.

These objectives were carefully considered in every phase of the GRODY project. Some of the objectives influenced decisions made during the design phase as well as the approach taken during the design phase.

1.3 GENERAL DESCRIPTION OF A DYNAMICS SIMULATOR

A general description of a dynamics simulator is included here to acquaint the reader with the type of design problem that confronted both development teams. A general knowledge of the type of problem becomes important later when the advantages and shortcomings of the different design methodologies are discussed.

The purpose of a dynamics simulator is to test and evaluate the onboard attitude control logic under conditions that simulate the expected in-flight environment as much as possible. The simulator can be considered a control system problem, beginning with an onboard computer (OBC) model that uses sensor data to compute an estimated attitude. Control laws are then modeled to generate commands to the attitude hardware (actuators) to reduce the attitude error. A truth model portion of the simulator simulates the response of the attitude hardware and generates a true attitude for the spacecraft. Sensor data corresponding to the true attitude are produced by the truth model and sent back to the OBC model (Figure 1-2).

1.4 THE TWO DESIGN TEAMS

The two design teams were approximately the same size, with 7 people on the Ada team and from 7 to 10 (average 9) on the FORTRAN team. The experience of the two teams was, however, somewhat different. The Ada team were more experienced in general, with more years of software development experience and a wider range of application experience. In addition, they were familiar with more programming languages, an average of seven compared to three for the FORTRAN team. On the other hand, the FORTRAN team were more experienced in the development of dynamics simulators. About two-thirds of the FORTRAN team had previously developed a dynamics simulator compared to only about two-fifths of the Ada team (Table 1-2).

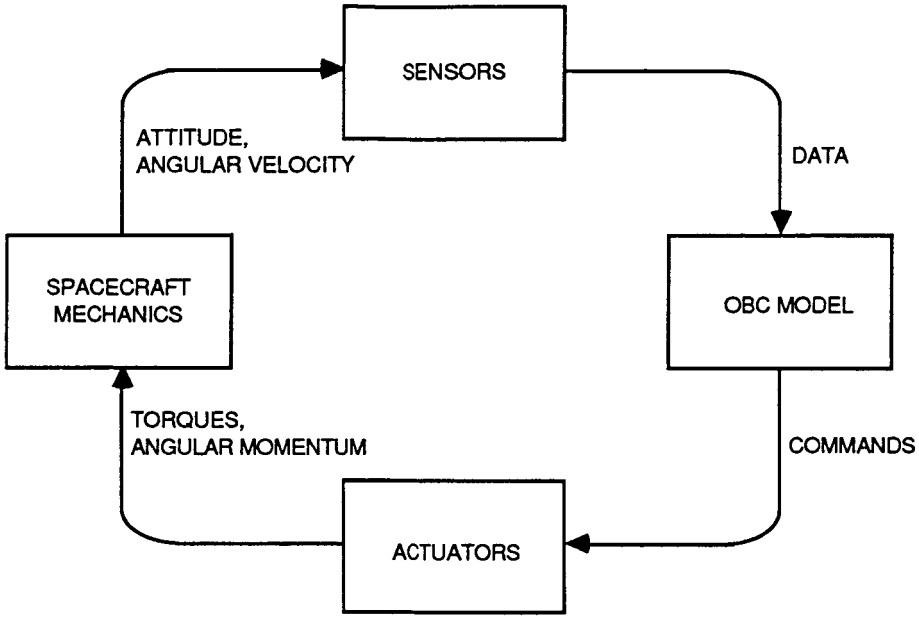


Figure 1-2. A Dynamics Simulator

Table 1-2. Team Profiles

CHARACTERISTIC	FORTRAN TEAM	ADA TEAM
NUMBER OF LANGUAGES KNOWN (MEDIAN)	3	7
TYPES OF APPLICATION EXPERIENCE (MEDIAN)	3	4
YEARS OF SOFTWARE DEVELOPMENT EXPERIENCE (MEAN)	4.8	8.6
TEAM MEMBERS WITH DYNAMICS SIMULATOR EXPERIENCE	66%	43%

0448 A4/4/7/87

1.5 DESIGN PROCESS WITHIN THE STANDARD SOFTWARE DEVELOPMENT LIFE CYCLE

The standard software development life cycle used in the flight dynamics environment is described in Reference 1. A brief description of this life cycle and the products that are usually generated during each phase are presented here so that comparisons between the Ada and FORTRAN life cycles can be made. The life cycle described here was generated for use on software development projects using FORTRAN. The effect of using this life cycle when developing in Ada will be discussed later in this document.

The standard life cycle can be divided into the following seven sequential phases:

- Requirements Analysis--During this phase, the developer analyzes a document that contains the functional specifications and requirements to assess the completeness and feasibility of the requirements and to make an initial estimate of the required resources. The results of this analysis are summarized in a requirements analysis report.

- Preliminary Design--In this phase, the design process is begun by organizing the requirements into functional capabilities and distributing these into subsystems.

- Detailed Design--In this phase, the design that was outlined during the preliminary design phase is expanded to describe all aspects of the system.

- Implementation--This phase consists of coding new modules from the design specifications, revising old code to meet new requirements, and unit-testing to ensure that each module functions properly.

- System Testing--During this phase, the completely integrated system produced during the implementation phase is tested according to a test plan (also generated during

the implementation phase) to verify that all required system capabilities function properly.

- Acceptance Testing--The testing during this phase is done by an independent team to ensure that the system meets all requirements.

- Maintenance and Operation--At this point, the software becomes the responsibility of a maintenance and operations group who implement any further enhancements and any error corrections that might be necessary.

The actual design process begins with a top-down approach to decompose the requirements. During the preliminary design phase, the development team organize the requirements into functional capabilities and then specify the major functional subsystems and their input/output interfaces and processing modes. The design is refined to a hierarchical level of two levels below the subsystem driver. During this phase, an initial determination is made of the available reusable code. This functional design of the system is documented in the preliminary design report and is presented for review in a preliminary design review (PDR). Responses to comments and criticisms received at the PDR are incorporated into the functional design contained in the final preliminary design report. This phase typically requires 10 percent of the time and 10 to 15 percent of the total effort required for the entire development cycle in a FORTRAN implementation (Reference 2).

During the detailed design phase, the functional design generated during the preliminary design phase is expanded to produce "code-to" specifications for the system. These include functional and procedural descriptions of the system, data flow descriptions, complete input/output file descriptions, operational procedures, descriptions of each module, and descriptions of all internal interfaces between modules.

Following the pattern of the preliminary design phase, these design details are documented in a detailed design document and presented in a critical design review (CDR). Again, responses to the comments received at the CDR are incorporated into the detailed design contained in the final detailed design document. This phase typically requires 15 percent of the time and effort required for the entire development cycle in a FORTRAN implementation.

SECTION 2 - DESIGN PROCESS APPROACH

As noted earlier, the FORTRAN team was able to move into the design phase of the software life cycle immediately after the requirements analysis phase, following the standard pattern for software development in the flight dynamics environment. A specifications and requirements document (Reference 5) provided the functional requirements and actually contained the highest level design for the FORTRAN development; the document is organized into major subsystems corresponding to the partitioning used for the last several successful simulator projects. The FORTRAN team was able to begin the design process by using this subsystem partitioning and then refining the design to include the lower level routines. Following this design pattern enabled the FORTRAN team to make the most use of code used successfully for other simulator projects, since the overall design of the FORTRAN system is similar to previous simulators. It also had the advantage of clarifying the interfaces between subsystems early in the project.

In contrast, the Ada team needed to begin their design process with a period of training, including training in design methodologies. The Ada team also discovered that several design issues began to surface during the requirements analysis phase, causing their requirements analysis phase to include some activities that are different from those usually undertaken during that phase.

2.1 DESIGN CONSIDERATIONS

2.1.1 FORTRAN DESIGN DRIVERS

From the beginning of the design phase, several factors were very influential in the development of the FORTRAN design. First, there was a large body of existing code for dynamics simulators that could possibly be reused--thus saving

development time and cost--if the design for the simulator was similar to previous ones. In addition, the FORTRAN team developing the design were very experienced with previous dynamics simulators and were familiar with the legacy of design success using particular interfaces and subsystem partitioning. The requirements document was organized into sections consistent with the previously successful subsystem partitioning, which further encouraged the FORTRAN team to reuse the previous design patterns. Finally, the FORTRAN team had more schedule pressure than the Ada team because the FORTRAN system was considered the real, operational software and thus needed to be ready to support the mission on schedule.

2.1.2 ADA DESIGN DRIVERS

One goal of the Ada team was to develop a dynamics simulator design that would take full advantage of the features Ada offers. To accomplish this, the Ada team needed thorough training in Ada and the Ada design methods, that is, those methods that encourage the full use of Ada's features. Although the Ada team consisted of very experienced developers, who were familiar with a wide variety of languages, Ada and its design methods were new to them.

Because there was no existing code in Ada that could be used, there was no tendency to adopt a particular design solely to maximize the amount of reusable code. Reusability was, however, still a factor influencing the design because another goal was to produce a design that would encourage the development of modules that could be reused in future Ada development efforts.

The Ada team had a strong desire to develop an independent design, one that was not influenced by the design of previous dynamics simulators. To do this, they needed the opportunity to work directly from the system requirements,

and it was important that the system requirements be language independent. One factor in favor of an independent design from the Ada team was their inexperience with the specific task of developing a dynamics simulator.

Finally, the Ada team needed time if they were to develop this new, independent design. Although the Ada team had a specific development schedule, they were not constrained by this schedule as were the FORTRAN team. They had the luxury of spending some extra time in areas where it was needed, such as in training, experimenting with different methodologies, and developing new methodologies when none of the existing ones seemed suitable.

2.2 ADA TEAM TRAINING

Because the Ada team members were new to Ada and its design methods, their software development cycle began with a training phase. The training phase was carefully designed to give the team a good working knowledge of Ada and its features and to acquaint them with several useful design methodologies for applications to be implemented in Ada. Two months of full-time effort were devoted to training each member of the Ada team, with the effort spread over a 6-month period (Table 2-1).

The training plan was carefully formulated by the experiment participants from the University of Maryland and consisted of several different types of activities. The resources used included an Ada textbook, an Ada language reference manual, and videotapes on the specifics of Ada. The videotapes were viewed in group sessions, followed by a discussion period, and then enhanced by reading and coding assignments. Toward the end of this language training, lectures on Ada-related design methods were presented. During these lectures, emphasis was placed on learning Grady Booch's Object-Oriented Design (Section 2.4.1), George Cherry's

Process Abstract Methodology for Embedded Large Applications (Section 2.4.2), and general software engineering methodology as well.

Table 2-1. Project Effort[a] Comparisons

PHASE	STAFF-HOURS		DURATION IN MONTHS	
	FORTRAN	ADA	FORTRAN	ADA
TRAINING	0	3225	0.0	6.0
REQUIREMENTS ANALYSIS	972	1393	1.5	2.0
DESIGN[b]	3227	3881	4.0	6.0
CODE/TEST	4734	10372[c]	6.0	16.0
SYSTEM TEST	2955	d	5.0	N/A
ACCEPTANCE TEST	2170	d	5.0	N/A

0448 A4/4/8/87

[a]EFFORT IS SUM OF TECHNICAL, MANAGEMENT, AND SUPPORT HOURS REPORTED ON SEL RESOURCE SUMMARY FORMS.
[b]HOURS UP TO CDR.
[c]ACTUAL HOURS THROUGH JUNE 1, 1987; ESTIMATED FOR JUNE 1 TO JULY 1.
[d]EXPECTED MAJOR SAVINGS WITH ADA.

The final training activity consisted of the design and implementation of a practice problem. This training problem was a team effort and consisted of nearly 6000 lines of code. More detailed information on the Ada training program and recommendations for the design of future Ada training programs are presented in References 6 and 7.

2.3 EFFECT OF REQUIREMENTS DOCUMENT ON DESIGN

The Ada team tried to begin the requirements analysis phase using the same approach taken by the FORTRAN team. They quickly realized that the requirements and specifications document (Reference 5) actually contained some of the high-level design used previously on dynamics simulator projects developed in FORTRAN. Because the team wanted to develop

an independent design, one that would be particularly suitable for an Ada development, they rewrote the specifications and requirements document using a specification approach called the Composite Specification Model (CSM) (Reference 8). Team members had been given an introductory lecture on CSM during their training and had no particular problem in using this method.

CSM allows a system to be represented from functional, dynamic, and contextual views. Using CSM as a specification tool provided information on its suitability for specifying requirements typical of those encountered in the flight dynamics environment. It also allowed the Ada team to become thoroughly familiar with the system requirements as they systematically analyzed them and reformulated them into the new specifications document (Reference 9). One feature of the new document was the description of functional processing in process specifications similar to PDL (program design language).

The team felt that the new specifications document successfully removed the bias toward a FORTRAN-like design by removing the inherited design features from the functional specification. They also felt they had gained a much better understanding of the system they were trying to develop. The requirements analysis report generated by the Ada team consisted of two parts: the rewritten requirements specification and a requirements analysis assessment report (Reference 10). The requirements assessment report detailed such areas as incomplete requirements, external data interfaces, existing code that might be reused, and initial resource estimates. The generation of these two documents completed the requirements analysis phase for the Ada team. During the requirements analysis phase, the Ada team spent 8.9 staff-months of effort over a 2-month period; the

FORTRAN team spent 6.2 staff-months of effort over a 6-week period (Table 2-1).

2.4 DESIGN APPROACHES INVESTIGATED

One of the objectives of the GRODY project was to investigate design methodologies that make effective use of Ada's features. Such methodologies use the so-called object-oriented approach, which means that objects are used as the basic unit of design instead of the traditional functional procedures. Previous FORTRAN designs used procedural abstraction rather than the object-oriented approach. In procedural abstraction, a particular subroutine can be thought of as a black box that provides a certain function and produces a particular set of output values whenever it is provided with specific input values. For example, applying this to the dynamics simulator problem, the truth model can be considered a procedural abstraction that provides the function of computing the current attitude. Whenever it is given particular input values and actuator commands, it provides the sensor data that correspond to that attitude state.

Typically, there are also certain object-oriented elements in the FORTRAN design. For example, when the truth model function provides the sensor data, the lower level routines are organized by objects so that the sensor data provided by a particular sensor (such as a fixed-head star tracker) are modeled in a particular routine designed to provide just that data. Similarly, another routine would model the data from another type of sensor, such as a fine Sun sensor.

To use Ada and its particular features (e.g., packages, information hiding, and tasks) effectively, the Ada team wanted to use a methodology well suited for these features. They investigated several different methodologies and actually developed the top-level design for the dynamics simulator in three different object-oriented methodologies. Because of

time constraints, it was impossible to explore each of these methodologies in more depth. Also, because all the designs were being developed concurrently, it was extremely difficult for the team to keep the design approaches completely separate. Even considering this difficulty, some fairly strong conclusions were reached about each of the methodologies explored. The following subsections describe the different methodologies tried and the advantages and disadvantages of each.

2.4.1 BOOCH OBJECT-ORIENTED DESIGN

The first object-oriented methodology used to approach the design of the dynamics simulator was developed by Grady Booch and is described in Reference 11. The usual application of Booch's design calls for translation of a textual specification into the design, using the technique of underlining nouns and verbs in the specification. The nouns map into objects in the design, and the verbs map into object operations. Obviously, some discretion must be used when choosing the nouns and verbs to be mapped into the design.

The design notation uses rectangles to represent Ada packages and parallelograms to represent tasks. Within each of these figures, there are two types of "windows": small rectangular windows to show visible procedures, functions, or entities and rounded windows to show visible data types. Hidden code is represented by blobs placed inside the figures. Arrows leading from an area of code indicate that the code uses an operation or data type in another object. Figure 2-1 is an example of Booch's design notation as it was applied to part of the dynamics simulator.

The graphic notation of Booch's methodology is clear and very descriptive of system objects, their component objects, and the use of one object by another. It does not, however, show which specific object operation is used, and it has no

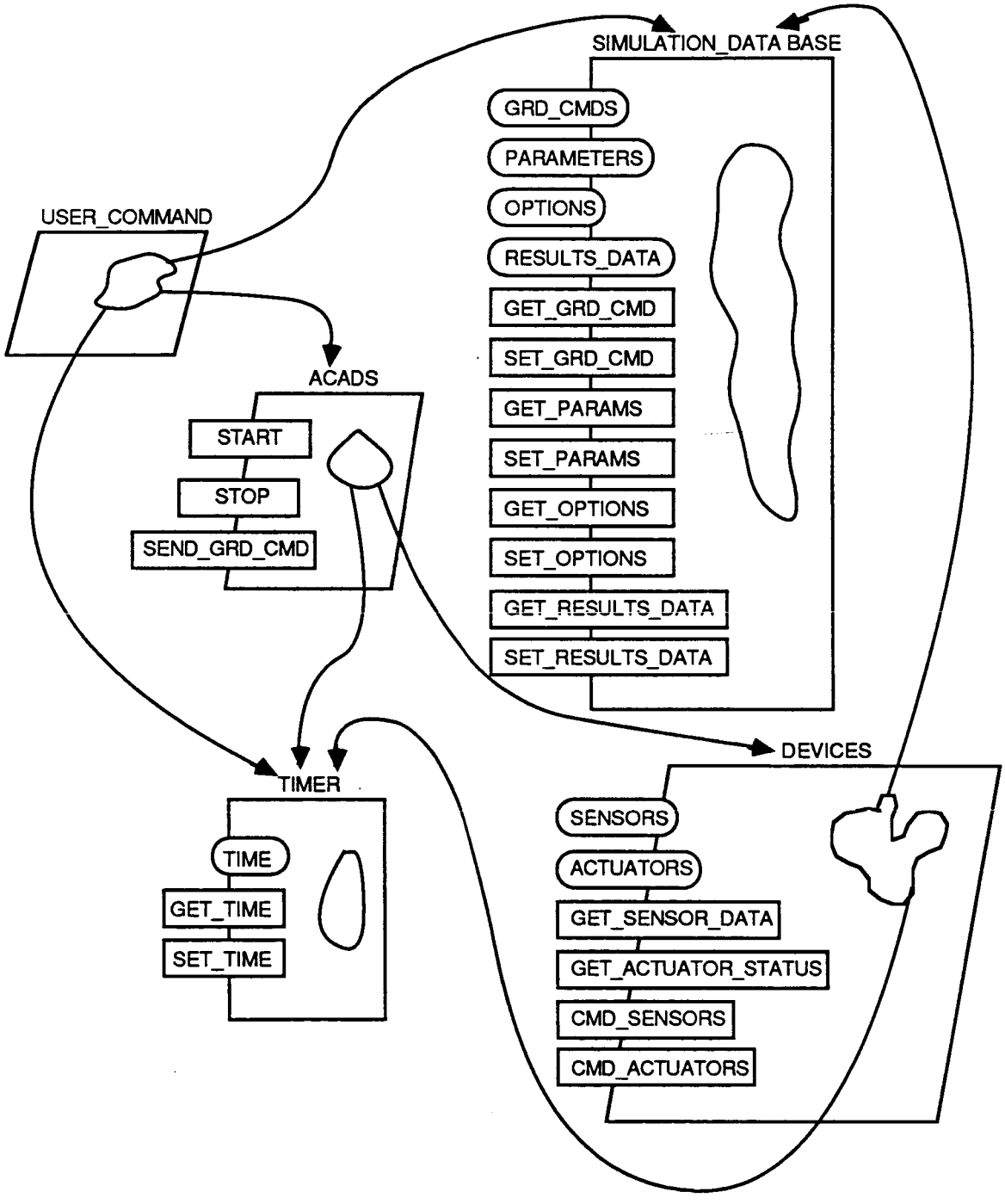


Figure 2-1. Booch Object-Oriented Design

method for showing data flows between objects. There is also no way to indicate a hierarchical structure, which is necessary for any large system.

The greatest drawback encountered with the Booch methodology was the technique for deriving the design from the specifications. The method of underlining text portions works well if the specification can be written in a few paragraphs, but it becomes a monumental task if the specification is of any size or complexity. In addition, the method does not provide any technique for deriving a design from a graphical specification.

2.4.2 PROCESS ABSTRACTION METHOD

Another methodology investigated by the Ada team was the Process Abstraction Method for Embedded Large Applications (PAMELA, or PAM for short). PAM was developed by George Cherry for use with real-time and embedded systems and is described in detail in Reference 12.

With PAM's design notation, processes are all concurrent objects and are represented by boxes. Arrows between the boxes represent rendezvous between processes. Labels on the arrows also provide a method for indicating data flow and some control information. Each process is marked either primitive (P) or nonprimitive (N). Primitive processes are Ada tasks, and nonprimitive processes are Ada packages that can be further decomposed until only primitive processes remain. Figure 2-2 is an example of this design notation. This decomposition allows a hierarchical structure to be represented using PAM, which is an advantage over Booch's methodology. The team also found that PAM provided a fair amount of guidance for constructing good processes.

PAM seemed to be very effective for the design of a real-time system. GRODY, however, was not specified as a real-time application with concurrent processes, even though parts of

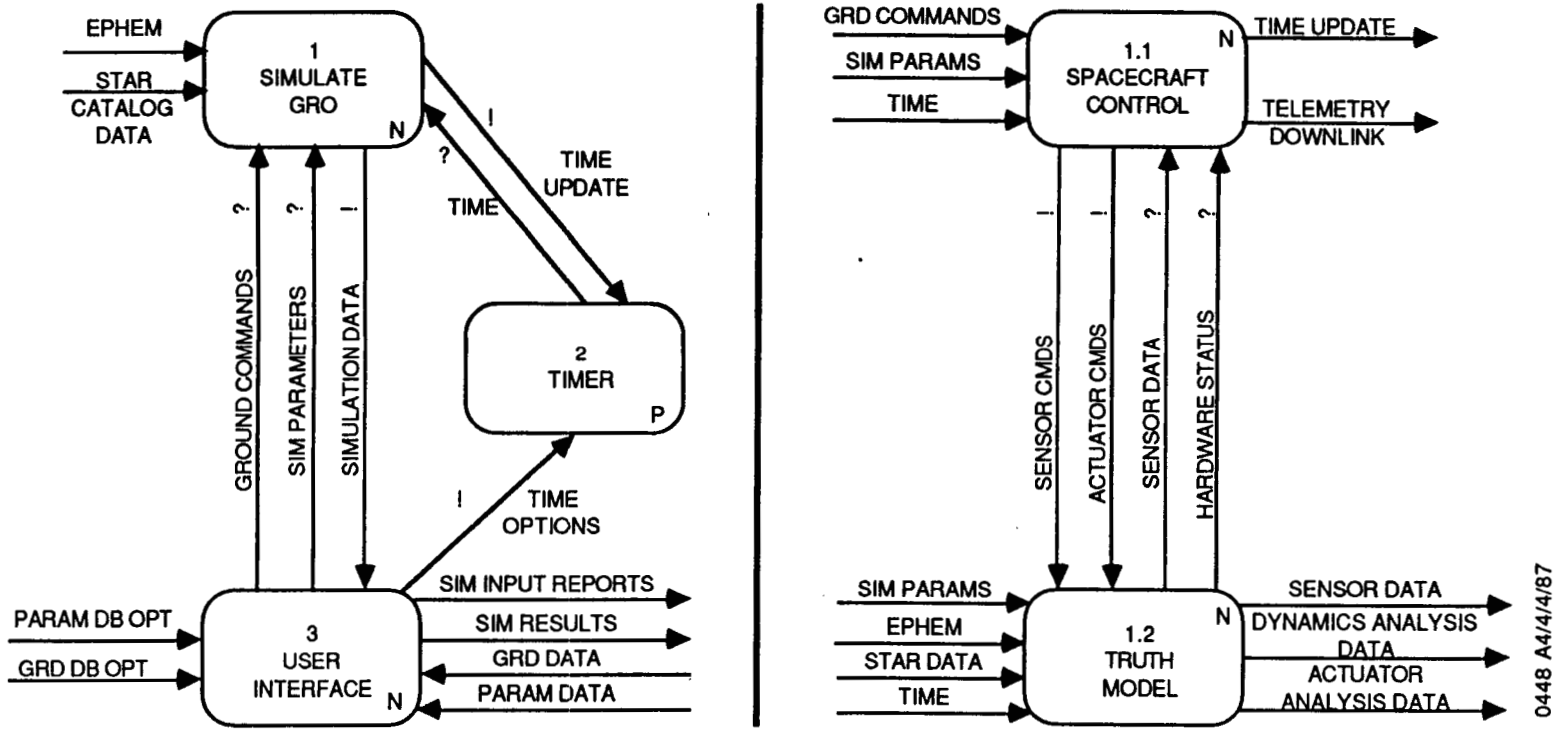


Figure 2-2. Process Abstraction Method

the system might be considered concurrent since GRODY models actions that occur in parallel in real life. Because the system is a simulation, it has no requirement for concurrent processes, except for the interaction the system has with the user. Thus, one interpretation of PAM's principles would leave large portions of the system represented as primitive processes with no method available for decomposing these into lower level entities. This problem limits the effective use of PAM for this type of application.

The only advantage in designing GRODY in terms of concurrent processes is the automatic scheduling of the sequence of process execution. This advantage is lost when considering the overhead of using Ada tasks and making Ada rendezvous. GRODY is required to run on a purely sequential machine and to be implemented in VAX Ada, in which this overhead can be quite large. In a simulator like GRODY, this consideration is particularly important because the main loop is a large portion of the system and may be executed tens of thousands of times per run. Obviously, such a performance degradation in this area could not be permitted.

2.4.3 OBJECT DIAGRAMS

The final methodology explored by the Ada team is one that the team had begun to develop during the design of the practice training problem. This approach tries to combine some of the best points of the previous design methodologies while expanding the approach so that it is flexible and general (References 13 and 14).

The notation of this object-oriented methodology has been named "object diagrams" and, as in the previous two notations, boxes represent objects. The hierarchical structure of the system can be shown by decomposing each object into lower level objects. Arrows on the object diagrams represent control flow or the use of an operation in the object

to which the arrow points. An object description lists operation names and the data flow between objects during each operation call.

Object diagrams thus illustrate the control structure of the system, and the object descriptions define the data flow in the various operations. The object diagrams are drawn with the senior-level controlling objects at the top of the page and the junior-level controlled objects below them. This serves as a convenient mechanism for demonstrating system hierarchy. As in PAM, each object can be decomposed into lower level component objects. Figure 2-3 represents an early version of the GRODY system using object diagrams.

The principles for constructing object diagrams are less explicit than those provided by the Booch methodology or PAM. They are based on the general principles of information hiding, abstraction, and design hierarchy. Efforts by the Ada team are in progress to outline these principles systematically to provide adequate guidance for designers attempting to use this methodology. Additional work is continuing to develop a technique for constructing object diagrams from a graphical specification (Reference 15).

2.5 CHOICE OF DESIGN METHOD

Of the three design methods explored during the preliminary design phase, only the object diagram methodology seemed complete enough to express the design of a complex system like a dynamics simulator. This methodology was therefore chosen to complete the preliminary design and to continue enhancing the design during the detailed design phase.

A few problems were noted with the early version of the GRODY design that was developed during the design methodology comparison period. The strong coupling between some of the objects is not effectively shown by the design diagram

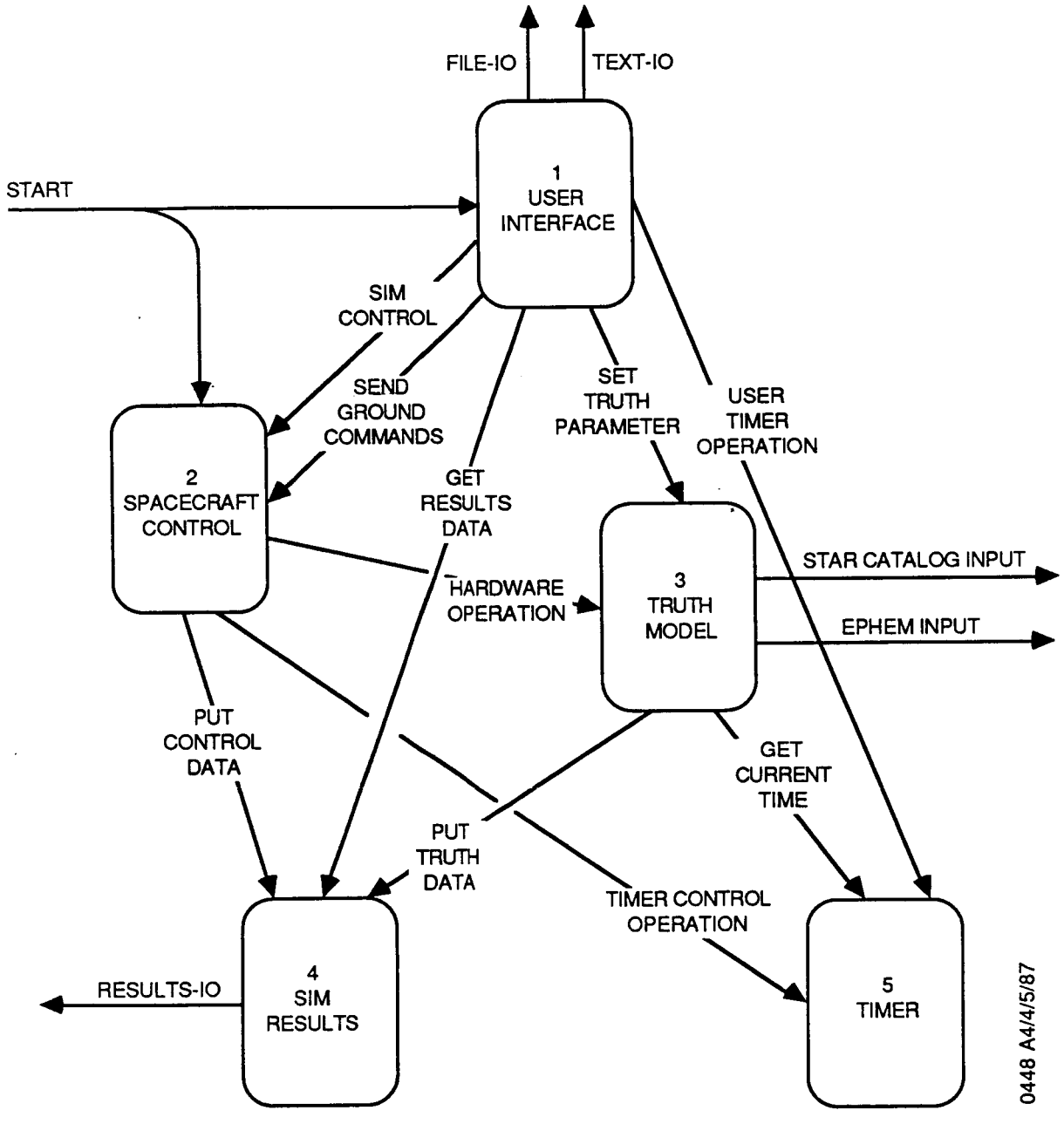


Figure 2-3. GRODY Team's Initial Design Using Their Own Methodology

in Figure 2-3. For example, there is a large set of parameters hidden in the truth model object that are used by the user interface object because the user needs to access these parameters, modify them, and store them. Modifications were made to the notation during preliminary design so that these relations could be shown. In fact, the design methodology evolved throughout the entire design phase as the Ada team found better ways to express the design they were developing.

2.6 DESIGN REVIEWS AND DESIGN PRODUCTS

The Ada team tried to follow the standard pattern in the flight dynamics environment concerning design reviews. This pattern calls for a PDR to be held at a stage in the development when the top levels of design are well defined, the requirements are divided into functional objects, and the interfaces between these objects are developed.

The FORTRAN team completed their preliminary design in about a month and a half and presented a PDR that included the usual design products for this phase. These included structure charts of the subsystems showing two levels of routines below the subsystem driver, data flow diagrams showing the interfaces between the subsystems, and the display formats. Prologs and PDL were available for the routines identified in the structure charts. Table 2-1 compares the length of the phases and the effort expended during these phases.

The Ada team had some difficulty deciding when they were actually ready to present the PDR because the clear guidelines applied for a FORTRAN development were not exactly applicable for an Ada development. A PDR (Reference 16) was held about a month after the design effort began with the rewritten specifications. The high-level design for GRODY was presented in terms of object diagrams and their accompanying operation dictionaries. As with the FORTRAN effort, the display formats were also presented. The team noted

that it was not meaningful (or even possible) to represent the Ada design with structure charts.

The detailed design phase lasted about 2 months for the FORTRAN team and culminated in the CDR. This review described the lower level routines, data flow, internal interfaces between modules, input and output files, and operational procedures. These were described in the design products of structure charts, data flow diagrams, file formats, and operational scenarios. In addition, all of this information was compiled into a detailed design document. Prologs and PDL were produced for all modules.

During the detailed design phase, the Ada team again had some difficulty in determining the appropriate time in development for the CDR. For example, the Ada team wanted to develop PDL in Ada, but this was very time consuming and tended to obscure the line between coding and design. After a 4-month period of detailed design work, the CDR was presented. Some team members felt that this was an arbitrary point in the design phase and that there was still a great deal of design work to be done. Others felt they were better prepared than usual because they had gained such a thorough understanding of the system they were developing.

The presentation at the CDR included object diagrams, operational scenarios, display formats, and sample PDL. The PDL used by the team was Ada-like but was not actually the compilable Ada they had hoped to use. A special reference guide explaining the object diagram symbols was distributed at the review to help attendees understand the object diagram notation. Between the PDR and the CDR, the object diagram methodology had continued to evolve, resulting in a new notation being used for the CDR and the detailed design documents.

The detailed design was recorded in a detailed design document completed shortly after the review. This document is being updated regularly during the implementation to ensure that it reflects the design actually being implemented.

SECTION 3 - LESSONS LEARNED

Now that the design phase of the GRODY project has been completed, it is possible to cite a number of areas where valuable experience has been gained concerning the use of Ada on applications software in the flight dynamics environment (Reference 17). A number of questions concerning the use of Ada for this type of application have at least preliminary answers that will be addressed in this section.

3.1 TRAINING AND EXPERIENCE

3.1.1 EFFECT OF TEAM EXPERIENCE ON THE DESIGN

Previous attempts to use Ada for scientific applications have usually resulted in an Ada system with a FORTRAN-like design (Reference 18). In the case of GRODY, the design generated by the Ada team looks very different from a FORTRAN design, attributed in part to several factors involving the team's experience.

First, because the Ada team were experienced in many languages, they were not set into any particular design pattern that might have been learned by using just one language. This broad base of experience enabled them to view the design problem from many different angles and to develop different approaches toward the design of the system.

Second, the Ada team were not experienced in using Ada, but they were very enthusiastic and eager to use all of Ada's advantages. This led to a sincere effort to develop an Ada-style design that would really test Ada's features in the flight dynamics environment.

Third, the Ada team were not experienced in designing dynamics simulators and thus were not biased by previous design efforts to generate dynamics simulators in FORTRAN. The team as a whole had a broad experience base free from

bias toward a particular language and a positive, enthusiastic attitude toward Ada.

3.1.2 EFFECT OF TRAINING ON DESIGN PRODUCTION

Obviously, when the goal is to produce a design using Ada's features to best advantage, it is essential that the design team be well acquainted with those features. As previously mentioned, the Ada team undertook a program of intensive Ada training before beginning the actual design work on the dynamics simulator. The training included a training problem that proved extremely useful.

To converge on an appropriate design, it is also essential that the team knows different design methods. Most programmer/designers in the flight dynamics environment use functional decomposition as their design method. Part of the training for the Ada team was in the use of other design methodologies. Cherry's PAMELA and Booch's object-oriented design methodologies are radically different from the standard procedural decomposition used in this environment. Such exposure was one source of broader insight into problem-solving for the team. To fully exploit Ada's features, various design methodologies, especially the one to be used for the project, must be included in the training; just knowing the language is not enough.

An appropriate design both exploits Ada's features and makes implementation easier, and the Ada team found that implementation was significantly promoted by their design. It was easy for a programmer to code from the design documents, even when the coder was not the designer for that section of the project. This has an important benefit in that it permits a buildup of staff during implementation, allowing parallel development. In a project with a tight schedule, managers may be able to increase the staffing to minimize time.

Additional training in Ada design methodology should be considered for those in managerial positions so that the managers will be able to interpret the design notation used during design reviews. Such an understanding will enable the managers to better assess the design being presented and to provide valuable feedback to the design team.

3.2 REQUIREMENTS AND SPECIFICATIONS

In the current flight dynamics environment, the specifications from which a development team works are heavily biased toward FORTRAN. In fact, the high-level design for the simulators is actually contained in the specifications document, and this design has not changed for several years. Therefore, to really explore the various design methodologies, the Ada team found they had to rewrite the specifications to remove the bias toward FORTRAN and the whole FORTRAN legacy. As previously mentioned, the requirements were rewritten using the Composite Specification Model.

It was during this respecification process that the highest level of the design began to take shape. Because the problem domain lends itself well to an object-oriented view, problem-solving proceeded along this line.

Team members felt that the resulting specifications were language neutral. The team had not yet had extensive experience with Ada, and this particular specification method existed before Ada. New specifications freed the team from the FORTRAN-oriented design built into the original specifications. One person felt that even the new specifications had a design bias built in; however, this bias was toward an object-oriented approach, and it was felt that this would not limit the development with Ada.

The team felt that rewriting the specifications increased their understanding of the problem more than merely analyzing the original specifications would have done. One

additional consequence of rewriting the specifications was that the team were prevented from postponing some important questions until implementation, which would have meant major design changes at that point.

Future projects in which Ada is being considered as the development language should concentrate more effort on the original specifications document. Ideally, the requirements should be structured in a language-independent form to allow the developers more latitude in choosing a design that satisfies the requirements and makes full use of the features of the implementation language.

It seems clear that new Ada developments will require more time during the requirements analysis, specification, and design phases. This extra effort should, however, result in a deeper understanding of both the problem and solution domains, yielding a higher quality product, better documentation of the earlier phases, and a cost savings during testing and maintenance.

3.3 ADA DESIGN METHODOLOGY

3.3.1 EARLY SELECTION OF DESIGN METHODOLOGY

The chances of producing a good Ada design can be greatly improved by selecting a design methodology that is appropriate for the type of project and by making this selection as early as possible. Selecting a methodology that is unsuitable for producing a complete design can result in considerable loss of design time. The Ada team spent a significant amount of time in developing their own methodology, which proved quite satisfactory for GRODY, but the development of this methodology extended their design time because they were essentially developing the design for both the project and the methodology concurrently.

3.3.2 EXPLOITATION OF ADA FEATURES THROUGH METHODOLOGY

If the methodology does not exploit Ada's features, why use Ada rather than another language? Many of Ada's benefits stem from the portability and maintainability gained by using packages, tasks, and generics--central features distinguishing Ada from most other languages.

One of the study objectives was to experiment with various design methodologies. The Ada team investigated structural decomposition, Cherry's PAMELA, and Booch's object-oriented design. They found that structural decomposition did not encourage the use of Ada's unique features; that PAMELA, which was designed for use with embedded systems, was too oriented toward concurrency for this application; and that Booch's object-oriented design methodology did not provide enough guidelines in its representations for a project of this size (it left too much up to the designer's judgment).

As a result, the team developed their own object-oriented methodology that incorporates ideas from both Cherry's and Booch's methods. The methodology produces object diagrams as the final result of object/data flow analysis. Two orthogonal hierarchies exist:

- Parent-child hierarchy (object decomposition)
- Seniority hierarchy (an object using the services of another is senior to the used object)

The new object-oriented methodology maps very well into Ada, because both were developed with modern software engineering concepts in mind (e.g., data abstraction, information hiding). Objects easily convert to packages, and packages encourage modularity.

One of the successful results from the design is the modularity. The team felt that this helped make interfaces easier to design, and increased interface reliability is

expected at testing. Another important effect of modularity in the design is the ease of adding new programmers to the project and phasing out others if required.

Another successful point is that the original design is still being followed in implementation, without major changes. The changes that have been made are additions. The team now feels that not enough attention was given to type specifications during design. However, it was felt that the object diagrams were helpful as a framework for discussing proposed changes.

3.4 DOCUMENTATION OF DESIGN

3.4.1 USEFULNESS OF OBJECT DIAGRAMS

Object diagrams are the key type of documentation produced by the Ada team's object-oriented methodology. Structure charts are the documentation produced with the standard FORTRAN design process.

The lack of a specific methodology at the start of the project was a problem, although unavoidable in this case because of the objectives of the study. The representations changed over time as the methodology developed, which was a big problem because it was difficult to keep the design documents consistent. To apply a methodology well, everyone needs to know the ground rules at the start. This facilitates understanding between developers on the team and between the team and the managers.

The key issue here is the importance of people's expectations. Less precision in structure charts and FORTRAN presentations at the PDR and CDR is acceptable than would be allowed with Ada documentation. Because the representations are so different for the Ada documentation, any unspoken understandings and intuition are lost.

Managers were unable to understand the object diagrams at these reviews. They tried to look at them as though they were the familiar structure charts, and could not visualize the design. Object diagrams contain a high level of detail, to express all the relationships they are capable of expressing. If some type of modification were made to suppress details of relationships between modules so that some relationships could be shown between a greater number of modules, the gap between object diagrams and structure charts would be lessened.

Even so, training is needed to make the object diagrams familiar to managers and reviewers. Unfamiliarity leads to concerns that something is being hidden. In addition, when the design is not understood because of the representation, the developers get less feedback on their design.

One clear implication of this experiment is the need for educating managers and reviewers in both Ada and the new concepts of software engineering. An Ada-oriented development requires a fair amount of knowledge on the part of the reviewers. More and different types of information must be examined to validate each phase of the life cycle.

3.4.2 COMPILABLE DESIGN ELEMENTS

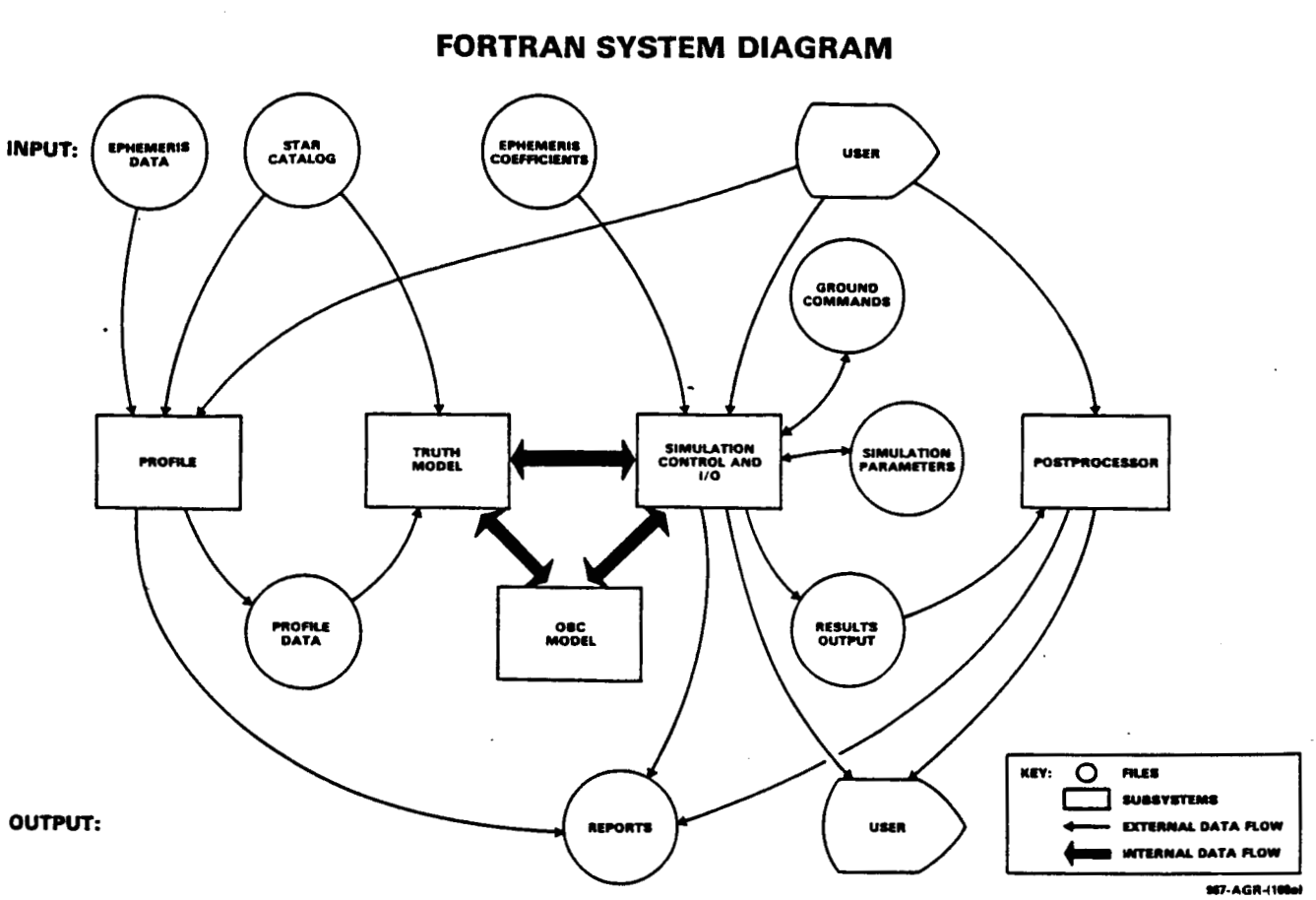
Another aspect of the design documentation that was investigated for GRODY is the concept of using compilable design elements. Ada itself can work well as a compilable PDL, whereas the PDL used with FORTRAN is pseudocode. The advantage of compilable PDL is that it permits interface checking and type checking, which help ensure the validity of the design in a way not otherwise possible at this early stage of development. This requires more precision in the design process than is required in the standard FORTRAN design process, but it provides more assurance and confidence during the PDR and CDR.

The Ada team developed only a small portion of the PDL for GRODY in compilable Ada elements during the design work that occurred before the reviews because they did not have enough time to do it all. However, they felt that this would have been very beneficial and actually did develop these elements in early implementation. Most team members felt that this activity should normally be considered a part of the design phase.

3.5 DIFFERENCES IN THE DESIGNS

The resulting FORTRAN and Ada designs were studied to determine if there were any real differences between them. Some previous experiences using Ada for scientific applications had shown that the design developed for the Ada system was very similar to a FORTRAN design (References 18 and 19). Since considerable effort has been expended during the GRODY project to develop an independent design, the question is--Is the design really different?

An examination of the two designs reveals several differences (Figure 3-1). The Ada design does "look" different from the FORTRAN design. The Ada design consists of one program with five subsystems, whereas the FORTRAN design involves three programs: a profile program, which calculates the attitude; the simulator, which consists of the truth model, the OBC model, and the simulation control logic; and the postprocessor program, which analyzes the results. The Ada program assigns different functional processing to its corresponding subsystems such that the functions of the FORTRAN profile program are incorporated into the Ada truth model. The Ada user interface performs the functions of the FORTRAN postprocessor and some of the user interaction performed by the FORTRAN simulation control program. The simulation support and control functions are separated into two Ada subsystems. The Ada OBC subsystem is



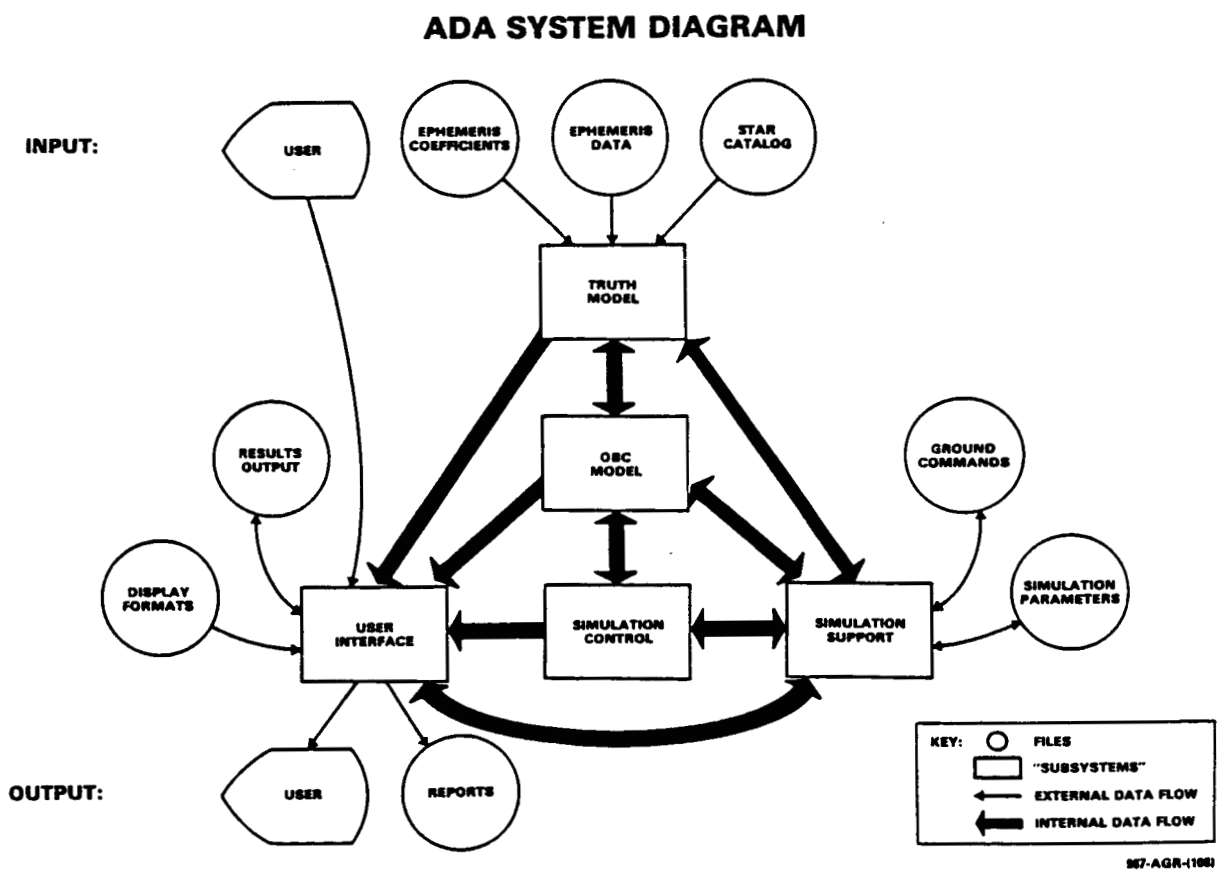


Figure 3-1. FORTRAN and Ada System Diagrams

functionally similar to the FORTRAN OBC subsystem. These differences in assignment of functional processing to the subsystems result in different data flows between the various subsystems.

In addition to these structural differences, there are two fundamental differences in the basic operation of the simulators. The first is that the FORTRAN simulator operates on a "fixed time step" principle, whereas the Ada simulator operates on a "calendar of events" principle. During an iteration of the FORTRAN simulator, the simulator control program wakes up the truth model, which computes the attitude state and places the corresponding sensor data in a holding area. The truth model then signals the control program to wake up the OBC subsystem, which obtains the sensor data, models the control laws, and generates the actuator commands, which are placed in a holding area for the truth model to use on the next iteration. The OBC then signals the control program to wake up the subsystem that writes out the analysis record resulting from this iteration. At the completion of this writing, control is returned to the simulator control program to begin the next iteration. In the FORTRAN design, the user sets the cycle time (the amount of time that the simulation clock is incremented), and this cycle time determines when events occur in the simulation. In the Ada design, there is an external timer that causes automatic advancement of the scheduler in the OBC so that the cycling of the clock is like that in the actual spacecraft OBC and is not under user control (Figure 3-2).

The second operational difference in the two designs is the passive nature of the truth model in the Ada design. The Ada OBC subsystem calls the truth model to obtain sensor data whenever it is needed. The user can control the cycle time in the truth model, but this does not affect the timing

in the OBC. More information on the design differences is presented in Reference 19.

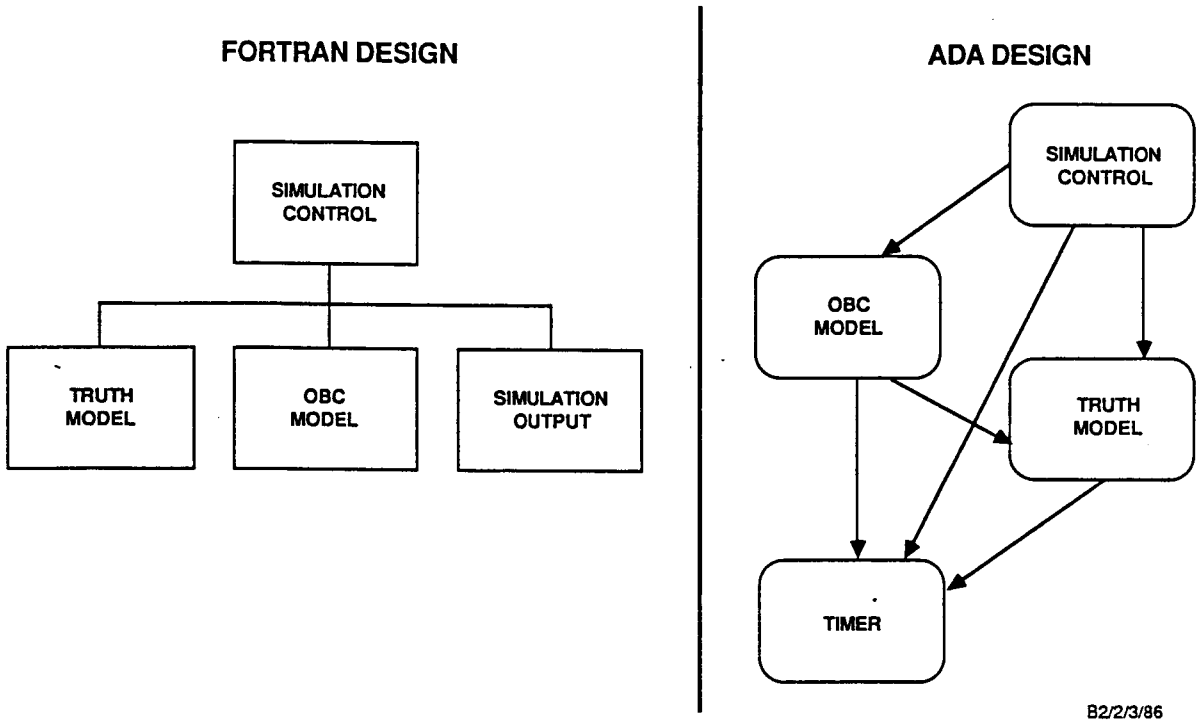


Figure 3-2. Comparison of FORTRAN and Ada Simulator Operations

3.6 CHANGES IN THE DESIGN PHASE OF THE LIFE CYCLE

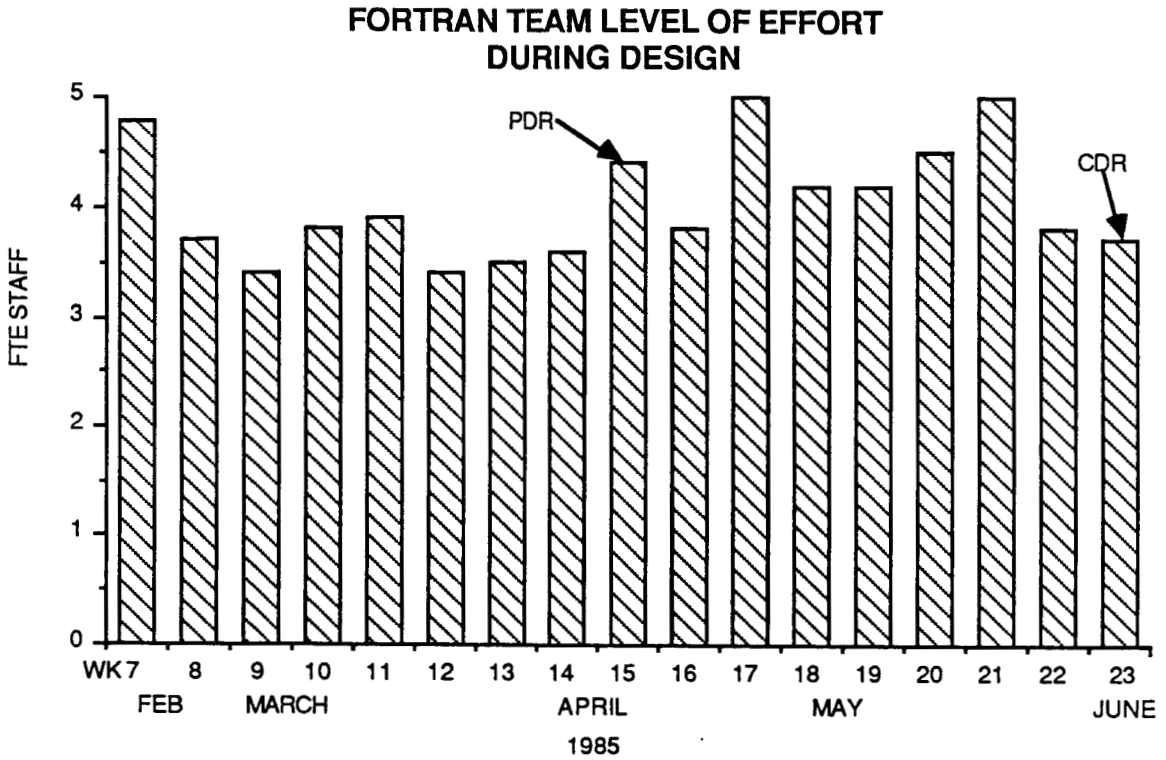
The legacy is that the starting point for design is a specifications document already containing the preliminary design. As has been shown, a preliminary design oriented toward FORTRAN would severely limit an Ada design because it would not take advantage of Ada's unique features. In this case, with the specifications rewritten, there was less design in the specifications document. However, since the document still contained some unfamiliar design, it is unclear exactly where requirements analysis of the specifications stopped and the design phase began.

The milestones of the design phase may also be different. In the usual software life cycle with FORTRAN, the requirements for a PDR and CDR are well defined, and the breaks between life-cycle phases seem logical and real. There is, however, no direct conversion for Ada, because the new object-oriented methodology and its documentation are so different from the traditional ones. Again, preliminary design seems to fade into detailed design, and detailed design fades into coding. It thus seemed that the PDR and CDR for the Ada design occurred at arbitrary times rather than at logical points in the design process.

Figure 3-3 shows the level of effort during design in weekly or biweekly increments. The Ada team level of effort shows large peaks around the time of the PDR and CDR, indicating that the team felt additional effort was necessary to prepare for these reviews. The FORTRAN team level of effort shows a similar peak for the PDR but a much more level curve approaching the CDR, with an actual decrease in design effort close to the CDR.

The members of the Ada team held different opinions on how well prepared they were for the PDR and CDR. One team member in particular felt more prepared than usual because he understood the design and its implications so well. Others felt less prepared than usual because of the newness of the methodology and representations and because they were unsure how to map the state of the design into the format generally expected at the PDR and CDR.

The concept of having two reviews seems to be desirable when designing in Ada, but it appears that the timing of these reviews should be different. For example, the PDR could occur later than normal, but with more rigor. The PDR could be represented by high-level compilable design elements, and



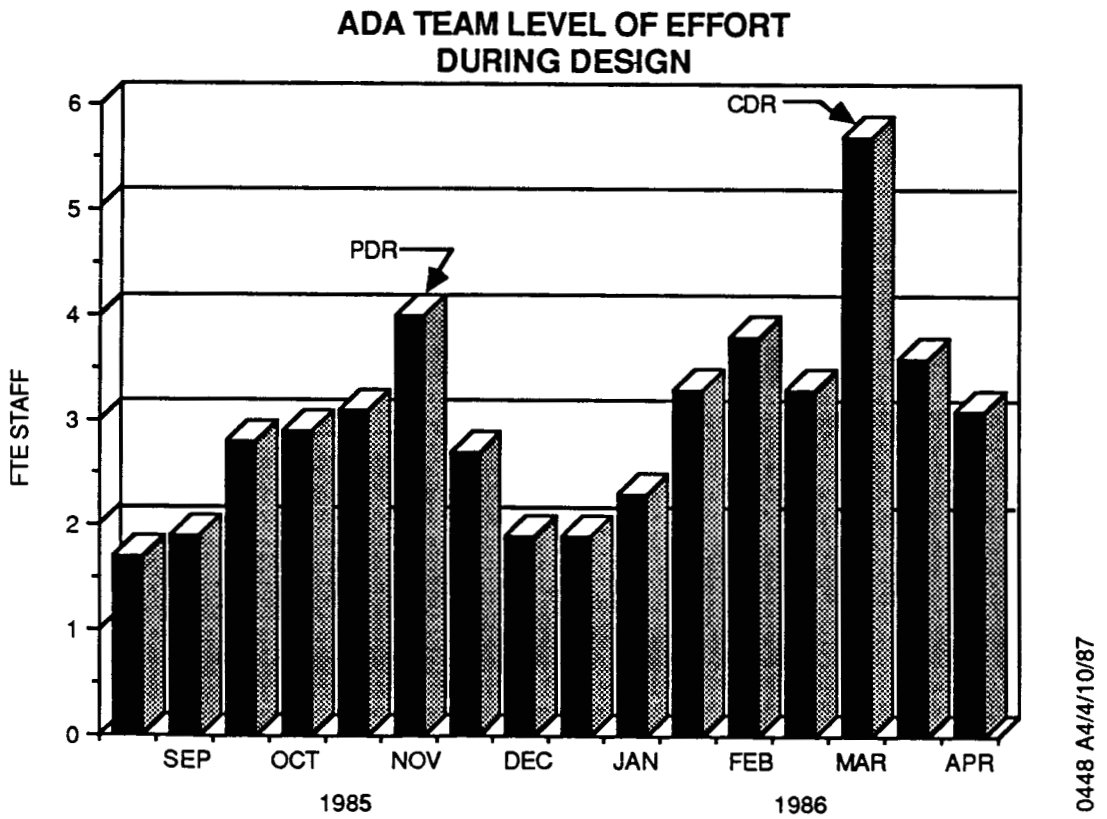


Figure 3-3. Levels of Effort for FORTRAN and Ada Teams During Design

CDRs might be staged for different design elements by examining more detailed Ada PDL pieces. The specific design products to be completed during each design phase should be defined for future Ada development projects. Figure 3-4 shows the differences in schedule between the two projects; requirements analysis, design, and implementation all took longer for the Ada project. Some Ada team members felt that design efforts continued into implementation and that a longer design phase would be helpful. It is hoped that the additional time spent in design will result in less time spent for the remaining phases.

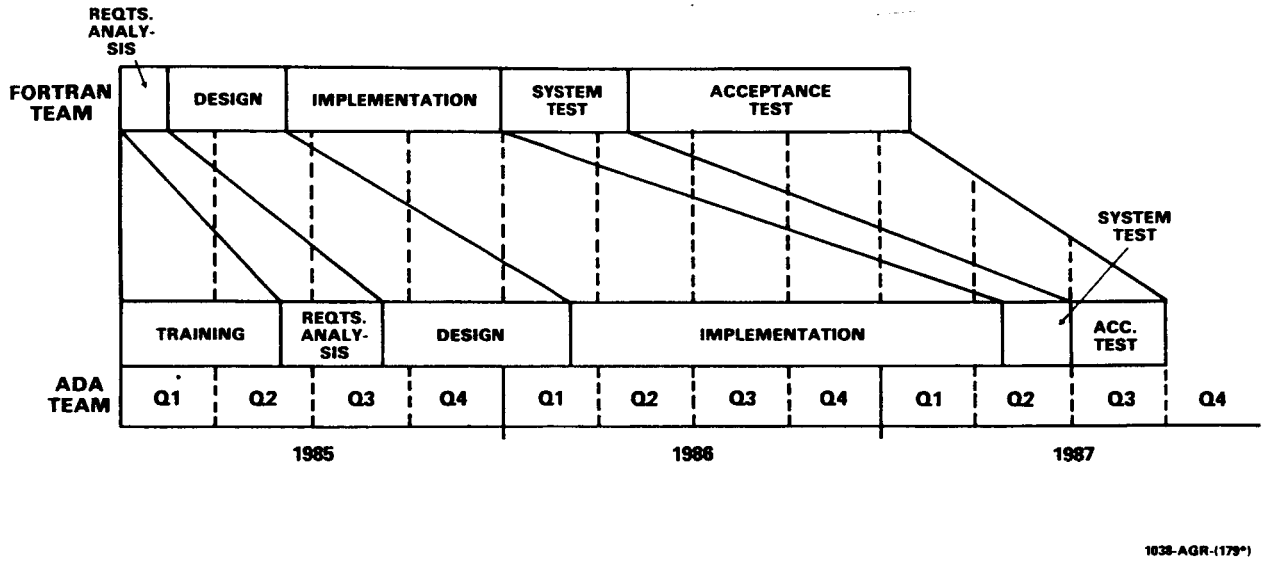


Figure 3-4. FORTRAN and Ada Team Schedules (Effort Levels Vary)

3.7 STAFFING CONSIDERATIONS

Because the development of GRODY was a new venture in the flight dynamics environment, the team assigned from the start of the project was the size team that would normally be used on this type of project during the implementation phase. This was done to train the team as a whole from the very beginning, since there was very little Ada expertise

available in this environment. From a training point of view, it was quite successful to have the whole team together from the initial phases of the project. However, during the requirements analysis phase, it was felt that a smaller initial team would have been able to perform more efficiently. The larger team size resulted in more time being spent in meetings.

The size of the team during the design phase was also larger than normal for the design of a project this size in the flight dynamics environment. This larger team was needed because the amount of design work done on GRODY was greater; the top-level design was actually done using three different methodologies. Even considering this, a future project might want to consider a larger design staff, since it seems to take longer to develop a complete Ada design. A complete Ada design does, however, describe the system better than a corresponding FORTRAN design because there are more expressible relationships (exception handling, etc.) and more design validation is possible (type checking, interface checking, etc.).

Future Ada development projects might benefit from using a smaller team initially and then building up during design and through implementation. This follows the type of staffing profile currently used on FORTRAN projects. As more people become familiar with Ada and its methodologies, it will be easier for them to join such a project during its later phases. As already noted, the object-oriented design produces a type of design that can be picked up and implemented by a coder who was not involved with the design.

3.8 COST OF USING NEW DESIGN

One important factor when considering a changeover to Ada and the new type of design that Ada implies is the real cost of such a change. In the flight dynamics environment, using

the type of design that takes full advantage of Ada's benefits means the loss of the whole FORTRAN legacy. This legacy includes old specifications, old design, old code, intuition, and institutional knowledge that is not recorded anywhere. Developing such a legacy for Ada takes time and costs money. Continuing observations of this project and future similar projects will provide more insight into the question of the value of Ada's benefits when compared to the cost of the changeover.

3.9 COST OF TRANSITIONING TO ADA AT DIFFERENT LIFE-CYCLE PHASES

Another factor influencing the cost of transitioning from FORTRAN to Ada is the point in the life cycle at which the change from FORTRAN development methods to Ada development methods occurs. The alternatives and their consequences are as follows:

Alternative	Consequence
Ada at project start	Best opportunity to cast requirements in a more language-neutral form
Ada after requirements analysis	Effective only if FORTRAN legacy is removed
Ada after design	Procedural "Adatran"
Source language conversion	Maintenance problems

This experiment chose to consider Ada as the development language from the beginning of the project. This seems to be the ideal situation because it provides the best opportunity to cast the requirements in a more language-neutral form. The developers are then free to develop the best possible design, based on both satisfaction of the requirements and efficient use of the development language.

The decision to use Ada could have been postponed until after a traditional requirements analysis phase. Based on

the experience gained during the GRODY project, such a decision would probably be effective only if the requirements had been written in a language-independent form before requirements analysis. Otherwise, the development team has two choices: First, they would need to spend additional time removing any previous language legacy from the requirements before any real design efforts could begin. Second, they would be constrained in their design choices by design features already ingrained in the requirements document. The first choice would result in a better Ada system but would cost more because of the additional time required to recast the requirements. The second choice would result in a design that might not be well suited for an Ada implementation and could result in a less reliable system that is more difficult to maintain.

If the change to Ada is made after the design phase is completed, the consequences are similar to the second choice but are even more pronounced. The design would certainly not be based on effective use of Ada's features and would not be able to use any of Ada's language-specific features. This would result in an "Adatran" program, that is, a program developed in Ada that looks just like a FORTRAN program. Such a program would lose any design advantages Ada might have offered, and the reliability and maintainability of the system could be affected. The cost of a conversion to Ada after design would thus be the increased cost of maintenance.

The last possibility to be considered is a decision to change to Ada after implementation in another language, probably FORTRAN. This would again result in an "Adatran" type of program that would be even less desirable than the previous case. When Ada is the original implementation language, certain design modifications or interpretations would probably be applied to make the design more suitable for

Ada. But in the case of source language conversion, none of this modification is practical, and the result would probably be the use of only a limited set of Ada's features. In addition, some of the features used might need to be molded to fit FORTRAN capabilities that did not quite correspond to the nearest Ada equivalent. Such changes would certainly result in a loss of reliability and an increase in maintenance problems. According to the SEL figures (Reference 2), the annual cost of error correction and maintenance usually ranges from 10 to 35 percent of the original development cost, so an increase in these percentages would be expensive.

Based on the experience gained during the GRODY project, it seems that the best time to transition to Ada is at the start of the project--even before the requirements and specifications are developed. This seems to be the only way to produce an Ada system that is efficient in its use of Ada and that can take advantage of Ada's features for increased portability and maintainability.

SECTION 4 - SUMMARY AND RECOMMENDATIONS

What have we learned from the design phase of GRODY, and where do we go from here? In response to the first question, the conclusions reached can be summarized as follows:

1. Training is important.
   a. A team designing a project to be implemented in Ada should be trained not only in Ada, but in several different design methodologies in addition to the one to be used on the project.
   b. Managers and reviewers should have some training in the design methodology to be used, to better evaluate the design and provide more useful feedback.
2. The specification method should not constrain the design. The requirements document should be language neutral and should not contain a bias toward any particular design method.
3. Methodology is important.
   a. The design methodology should be chosen as early as possible so that the team can be trained and valuable time will not be wasted trying to use an unsuitable methodology.
   b. The methodology chosen should exploit Ada's features (e.g., packages, task, and generics).
   c. Object diagram methodology seems to be an extremely useful method for developing the design for the type of project encountered in the flight dynamics environment.

4. Documentation of an Ada design requires different design products than are used to describe a FORTRAN design.

   a. Object diagrams were a very suitable representation for the GRODY design.

   b. Compilable design elements developed in Ada are very useful for providing validation of the design as well as for documentation.

5. Designing with Ada may imply different starting and ending points of the design phase.

6. It costs money to make a changeover to a new design and discard all the previous legacy associated with all phases of a FORTRAN development.

In response to the second question--Where do we go from here?--the following recommendations can be made for future projects. It seems wise to modify the usual software life cycle when developing with Ada by expanding the design phase to allow more time for design. The PDR and CDR should be retained but should occur at different points in the life cycle.

The PDR should occur later in the design phase and should include descriptions of the high-level elements and their input/output interfaces. These high-level elements could be well represented by using object diagrams. The description of these elements should be completed with compilable PDL. The CDR would then include a description of the more detailed design elements.

Continued study of GRODY and similar projects in the future will determine the suitability of Ada in the flight dynamics environment and will determine if the advantages gained by the use of Ada will outweigh the loss of the FORTRAN legacy.

GLOSSARY

CDR	critical design review
CSC	Computer Sciences Corporation
CSM	Composite Specification Model
GRO	Gamma Ray Observatory
GRODY	Gamma Ray Observatory Dynamics Simulator
GSFC	Goddard Space Flight Center
NASA	National Aeronautics and Space Administration
OBC	onboard computer
OOD	object-oriented design
PAM	Process Abstraction Method
PAMELA	Process Abstraction Method for Embedded Large Applications
PDL	program design language
PDR	preliminary design review
SEL	Software Engineering Laboratory
SLOC	source lines of code

REFERENCES

1. Software Engineering Laboratory, SEL-81-205, Recommended Approach to Software Development, F. McGarry, G. Page, S. Eslinger, et al., April 1983

2. Software Engineering Laboratory, SEL-84-001, Manager's Handbook for Software Development, W. Agresti, F. McGarry, D. Card, et al., April 1984

3. Goddard Space Flight Center, "An Experiment With Ada --The GRO Dynamics Simulator Project Plan," F. McGarry and R. Nelson, April 1985

4. Computer Sciences Corporation, PCA/IM-85/055(455), "The Ada Experiment--An Interim Report," W. Agresti, December 1985

5. Computer Sciences Corporation, CSC/SD-85/6106, Gamma Ray Observatory Dynamics Simulator Requirements and Mathematical Specifications, G. Coon, April 1985

6. Software Engineering Laboratory, SEL-85-002, Ada Training Evaluation and Recommendations, R. Murphy and M. Stark, October 1985

7. W. Agresti, "Measuring Ada as a Software Development Technology in the Software Engineering Laboratory (SEL)," Proceedings, Tenth Annual Software Engineering Workshop, NASA/GSFC, December 1985

8. Software Engineering Laboratory, SEL-87-003, Guidelines for Applying the Composite Specification Model (CSM), W. Agresti, June 1987

9. Computer Sciences Corporation, CSC/TM-85/6108, Specification of the GRO Dynamics Simulator in Ada (GRODY), W. Agresti, E. Brinker, P. Lo, et al., November 1985

10. Computer Sciences Corporation, PCA/IM-85/041(455), "GRO Dynamics Simulator (GRODY) Requirements Assessment Report," W. Agresti, E. Brinker, P. Lo, et al., September 1985

11. G. Booch, Software Engineering With Ada. Menlo Park, California: Benjamin/Cummings Publishing Co., Inc., 1983

12. G. W. Cherry, "Advanced Software Engineering With Ada--Process Abstraction Method for Embedded Large Applications," Language Automation Associates, Reston, Virginia, 1985

13. Goddard Space Flight Center, "Some Principles in Object-Oriented Design," E. Seidewitz, August 1985

14. Software Engineering Laboratory, SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

15. E. Seidewitz and M. Stark, "Toward a General Object-Oriented Software Development Method," Proceedings of the First International Symposium on Ada for the NASA Space Station, Houston, Texas, June 1986

16. Computer Sciences Corporation, PCA/IM-85/052(455), "GRO Dynamics Simulator in Ada (GRODY) Preliminary Design Report," W. Agresti, E. Brinker, P. Lo, et al., December 1985

17. C. Brophy, W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," Proceedings of the Washington Ada Symposium, Washington, D.C., March 1987

18. V. Basili et al., "Characterization of an Ada Software Development," Computer, September 1985, vol. 18, no. 9, pp. 53-65

19. W. Agresti, V. Church, D. Card, and P. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, Houston, Texas, June 1986

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, August 1983

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-406, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1986

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-86-006, Proceedings From the Eleventh Annual Software Engineering Workshop, December 1986

SEL-87-001, Product Assurance Policies and Procedures for Flight Dynamics Software Development, S. Perry et al., March 1987

SEL-87-002, Ada Style Guide (Version 1.1), E. Seidewitz et al., May 1987

SEL-87-003, Guidelines for Applying the Composite Specification Model (CSM), W. W. Agresti, June 1987

SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey, C. Brophy, et al., July 1987

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[4]Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

[3]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

NOTES:

[1]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4]This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.